Verbessertes Training von neuronalen Netzen

Vorlesung 3, Deep Learning

Dozenten: Prof. Dr. M. O. Franz, Prof. Dr. O. Dürr

HTWG Konstanz, Fakultät für Informatik

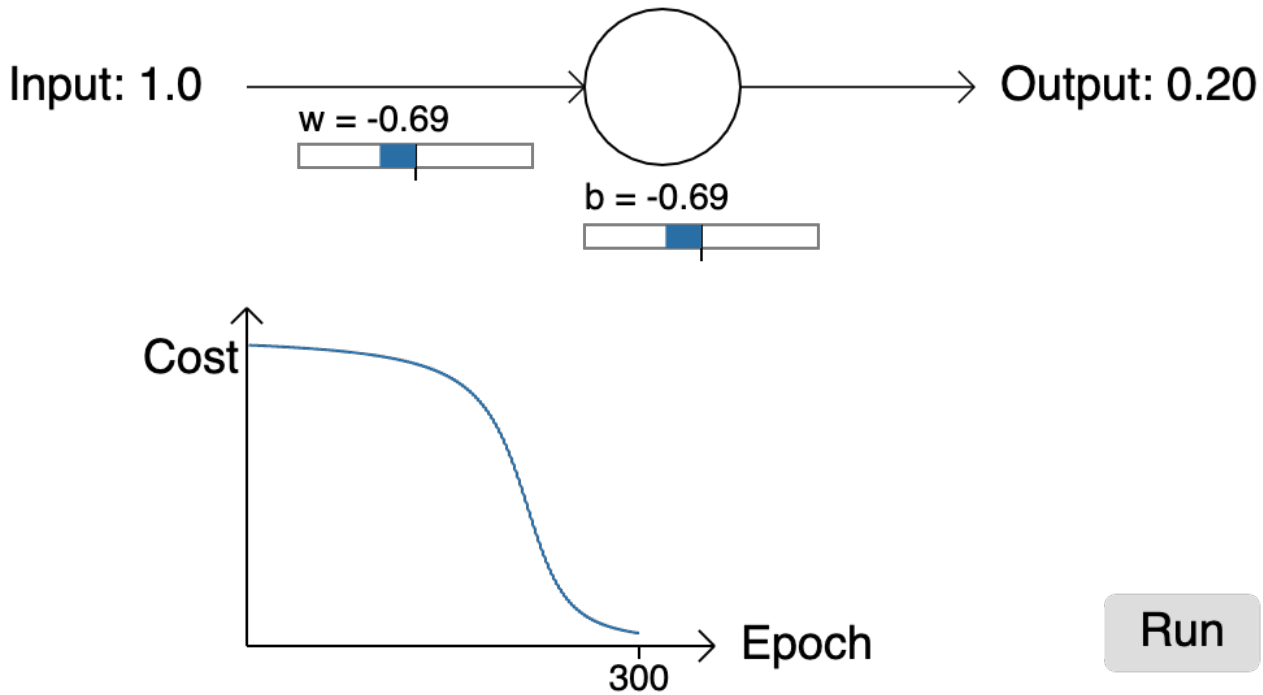
Übersicht

Übersicht

Verlangsamtes Lernen (Beispiel)

Sinnvoll wäre es, wenn ein neuronales Netz schnell lernt, wenn es große Fehler macht, und langsam bei kleinen Fehlern. Leider ist das in der Realität nicht der Fall.

Beispiel: einfaches Netz aus einem sigmoiden Neuron, quadratische Kostenfunktion. Input: immer 1, Label/Trainingsoutput: immer 0.



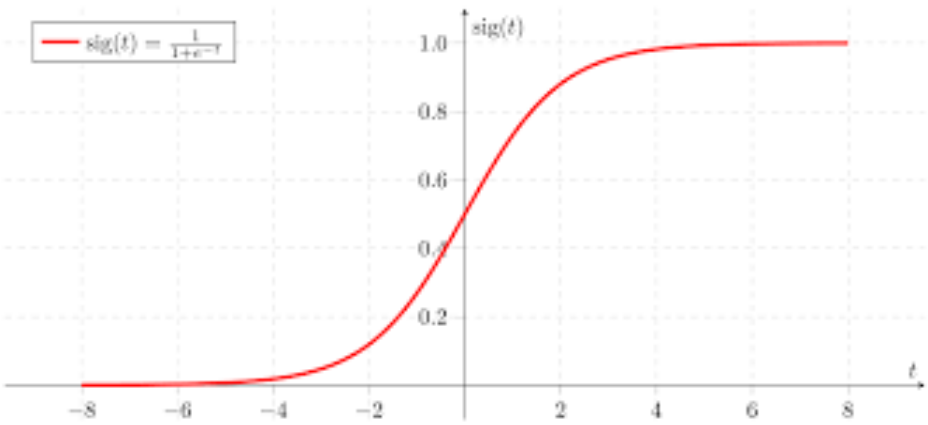
[Demo: https://neuralnetworksanddeeplearning.com/chap3.html]

Verlangsamtes Lernen: Gradient verschwindet in der Sättigung

Quadratische Kostenfunktion: $C = \frac{1}{2}(y-a)^2$, Output: $a = \sigma(z)$ mit $z = wx + b$.

Gradient:

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z) \quad \text{und} \quad \frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z)$$



[Wikipedia]

Wenn der Output des Neurons nahe an 1 ist, verschwindet die Ableitung von $\sigma(z)$ und damit der Gradient von w und b.

Verlangsamtes Lernen tritt auch bei deutlich komplexeren Netzwerken auf.

Kreuzentropie (1)

Bei Klassifikationsproblemen wird bei neuronalen Netzen für die Labels gern One-Hot-Coding verwendet: bei N Klassen besteht der Output des Netzes aus N Neuronen. Die zugehörigen Labels sind ebenfalls N-dimensionale Vektoren, wobei alle Einträge 0 sind, nur der Eintrag der korrekten Klasse ist 1.

Eine alternative Kostenfunktion für diesen Fall ist die Kreuzentropie (zunächst für ein Einzelneuron):

$$C = -\tfrac{1}{n} \sum_x \left[y \ln a + (1-y) \ln(1-a) \right].$$

C ist immer positiv, wenn a nahe y liegt, wird C minimal. Ableitung:

$$\frac{\partial C}{\partial w_j} = -\tfrac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \frac{\partial \sigma}{\partial w_j} = -\tfrac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \sigma'(z) x_j$$

$$\text{bzw.} \quad \frac{\partial C}{\partial w_j} = \tfrac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1-\sigma(z))} (\sigma(z) - y).$$

Kreuzentropie (2)

Für die Ableitung der Sigmoidfunktion $\sigma(z) = 1/(1+e^{-z})$ gilt

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)).$$

Damit vereinfacht sich der Gradient zu

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y),$$

d.h. $\sigma'(z)$ verschwindet, und der Lernfortschritt ist proportional zum Fehler $\sigma(z) - y$. Damit wird das verlangsamte Lernen von w vermieden!

Für b gilt ähnlich $\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y)$.[https://neuralnetworksanddeeplearning.com/chap3.html]

Erweiterung der Kreuzentropie auf mehrere Outputneuronen a_j:

$$C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right].$$

Softmax-Nichtlinearität

Statt der Sigmoidfunktion wird häufig die Softmax-Nichtlinearität als Aktivierungsfunktion in der letzten Schicht eingesetzt:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

Alle Aktivierungen sind immer positiv und summieren sich zu

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1,$$

d.h. der Output der Softmax-Schicht ist eine **Wahrscheinlichkeit**.

Damit kann die Aktivierung des j-ten Neurons als die vom Netzwerk geschätzte Wahrscheinlichkeit dafür interpretiert werden, dass die j-te Klasse vorliegt.

Log-Likelihood

Sind die Outputs eines Netzwerks Wahrscheinlichkeiten, so kann man die Log-Likelihood als weitere Kostenfunktion definieren:

$$C = \ln a_y^L.$$

Bedeutung: ist die wahre Klassenzugehörigkeit y, so ist die logarithmierte Wahrscheinlichkeit des Outputneurons für Klasse y ein Gütemaß für die Übereinstimmung zwischen Klassifikator und Label. Hier geht man also nicht von One-Hot-Coding aus, sondern von Klassenindizes als Labels.

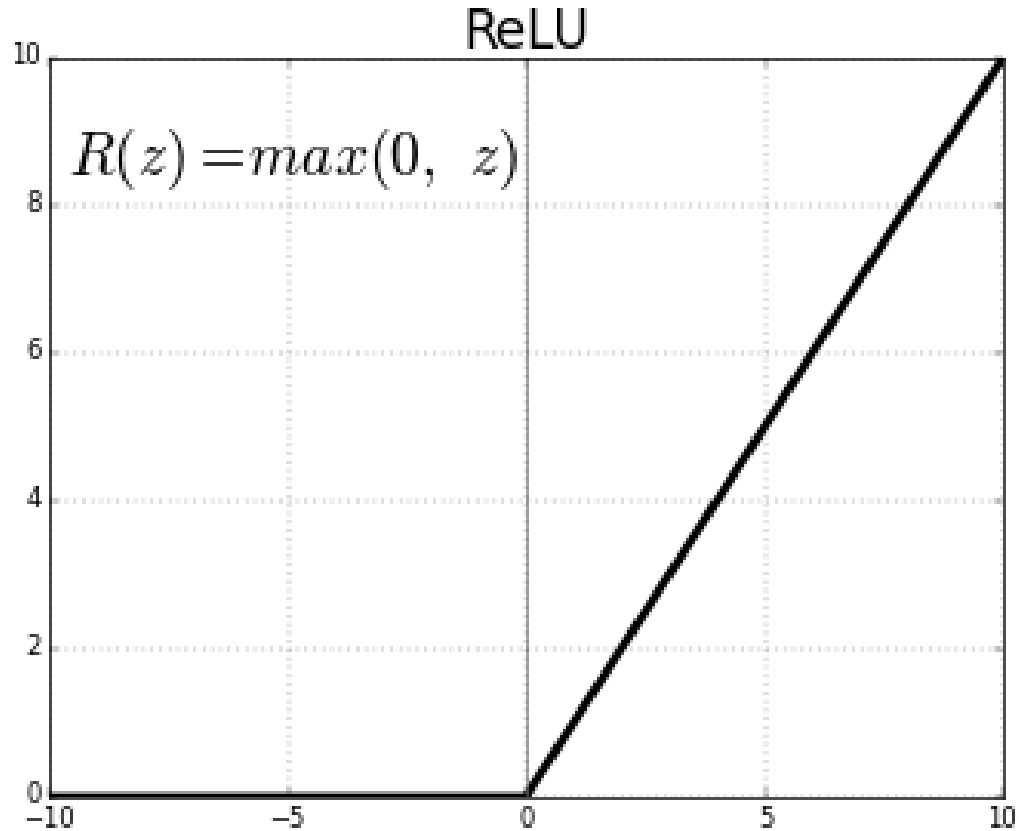
Ableitung von C für Softmax:

$$\begin{aligned} \frac{\partial C}{\partial b_j^L} &= a_j^L - y_j \\ \frac{\partial C}{\partial w_{jk}^L} &= a_k^{L-1}(a_j^L - y_j) \end{aligned}$$

Auch hier tritt das verlangsamte Lernen nicht auf!

ReLU: eine Nichtlinearität, die nie in Sättigung geht...

ReLU: Rectified Linear Unit



[Medium]

Wird sehr häufig (und erfolgreich) in den verdeckten Schichten verwendet, da diese Nichtlinearität nicht in Sättigung gehen kann und damit verlangsamtes Lernen vermieden wird.

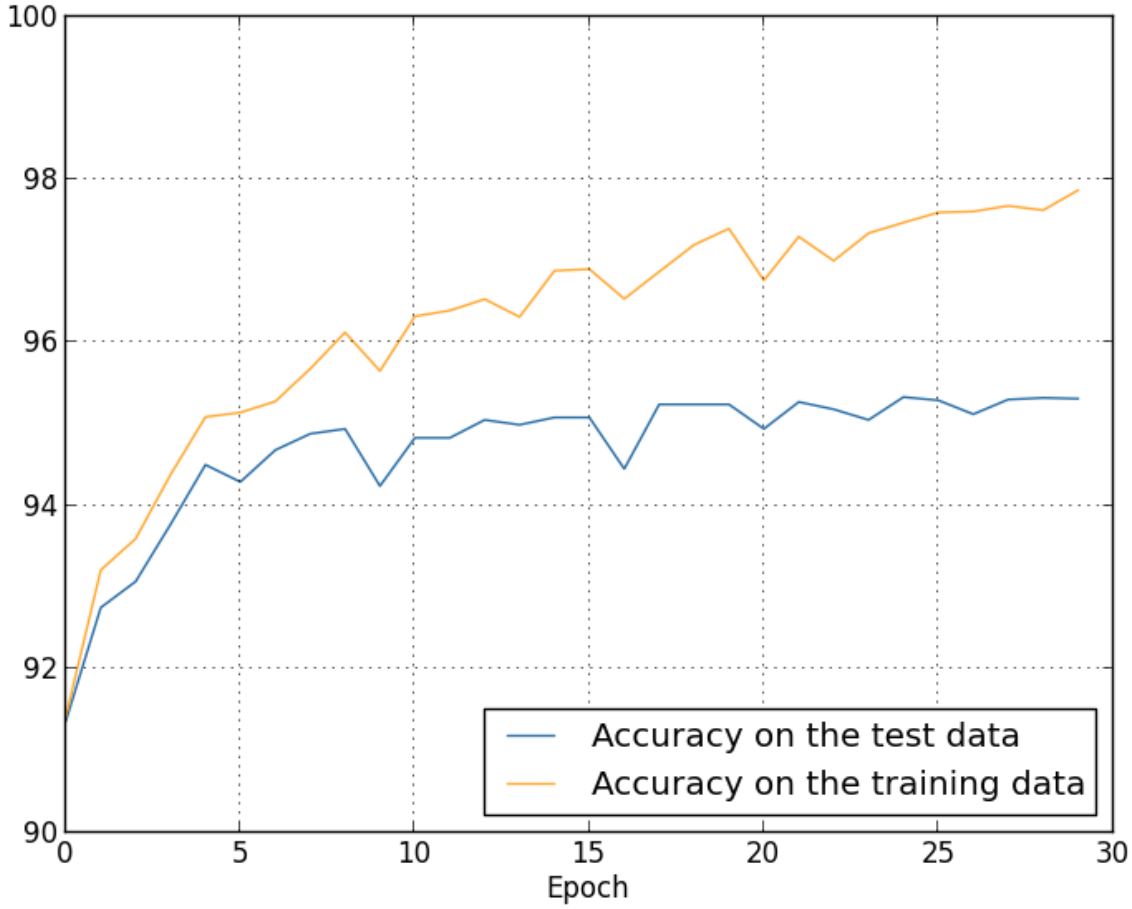
Fazit: Vermeidung des verlangsamten Lernens

Szenario	Nichtlinearität in der Ausgangsschicht	Kostenfunktion
Klassifikation, One-Hot-Coding, Klassen schließen sich gegenseitig nicht aus.	Sigmoidfunktion	Kreuzentropie
Klassifikation, Klassenindizes als Labels, Klassen schließen sich gegenseitig aus.	Softmax	Log-Likelihood
Regression	Linear	Mittlerer quadratischer Fehler

Für die verdeckten Schichten wird heute standardmäßig ReLU eingesetzt.

Übersicht

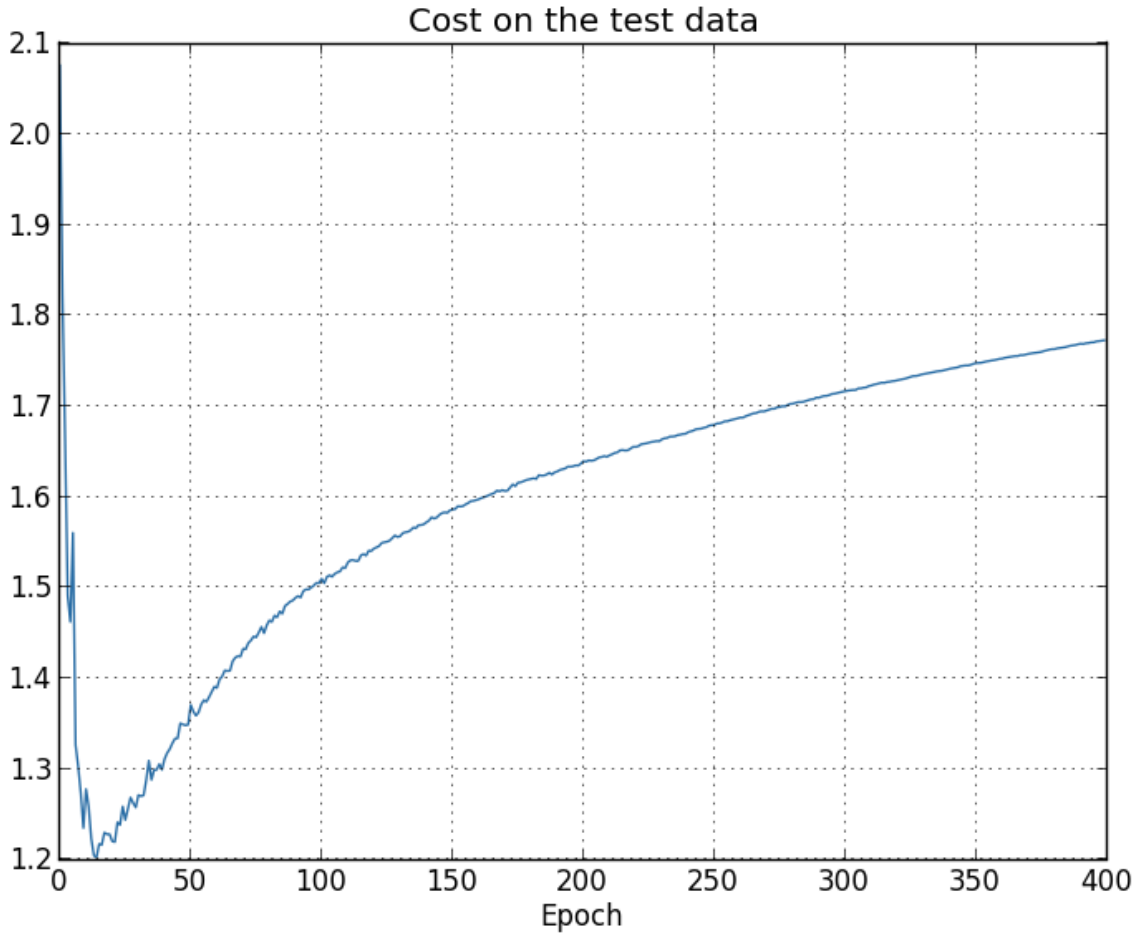
Experiment: Overfitting auf MNIST



[Nielsen 2016]

Overfitting: MLP passt sich an unbedeutende Details des Trainingsdatensatzes an, statt zu generalisieren. Nicht überraschend: wir passen ca. 24.000 Parameter an 1000 Datenpunkte an.

Early Stopping



[Nielsen 2016]

Einfache Strategie: Die Kostenfunktion auf dem Validierungsdatensatz wird regelmäßig während des Trainings berechnet. Sobald sie sich nicht mehr verbessert, wird das Training abgebrochen.

L_2-Regularisierung

- Das MLP ist immer noch ein Perzeptron, d.h. es minimiert nur eine Kostenfunktion in einem komplexen Merkmalsraum (berechnet von den verdeckten Schichten). Die Trennbreite der gefundenen Trennflächen spielt keine Rolle. Wie wir bei der SVM gesehen haben, ist die Trennbreite wichtig für die Generalisierung.
- Idee: ergänze die Kostenfunktion wie bei der SVM um einen Regularisierungsterm, der die Länge bzw. die L_2-Norm des Gewichtsvektors minimiert:

$$C = -\frac{1}{n} \sum_{xj} \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right] + \frac{\lambda}{2n} \sum_w w^2,$$

oder

$$C = \frac{1}{2n} \sum_x \|y - a^L\|^2 + \frac{\lambda}{2n} \sum_w w^2.$$

Ergebnis: Weight Decay

Regularisierte Kostenfunktion:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

Ableitung nach w (b wird meist nicht regularisiert):

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$

Lernregel:

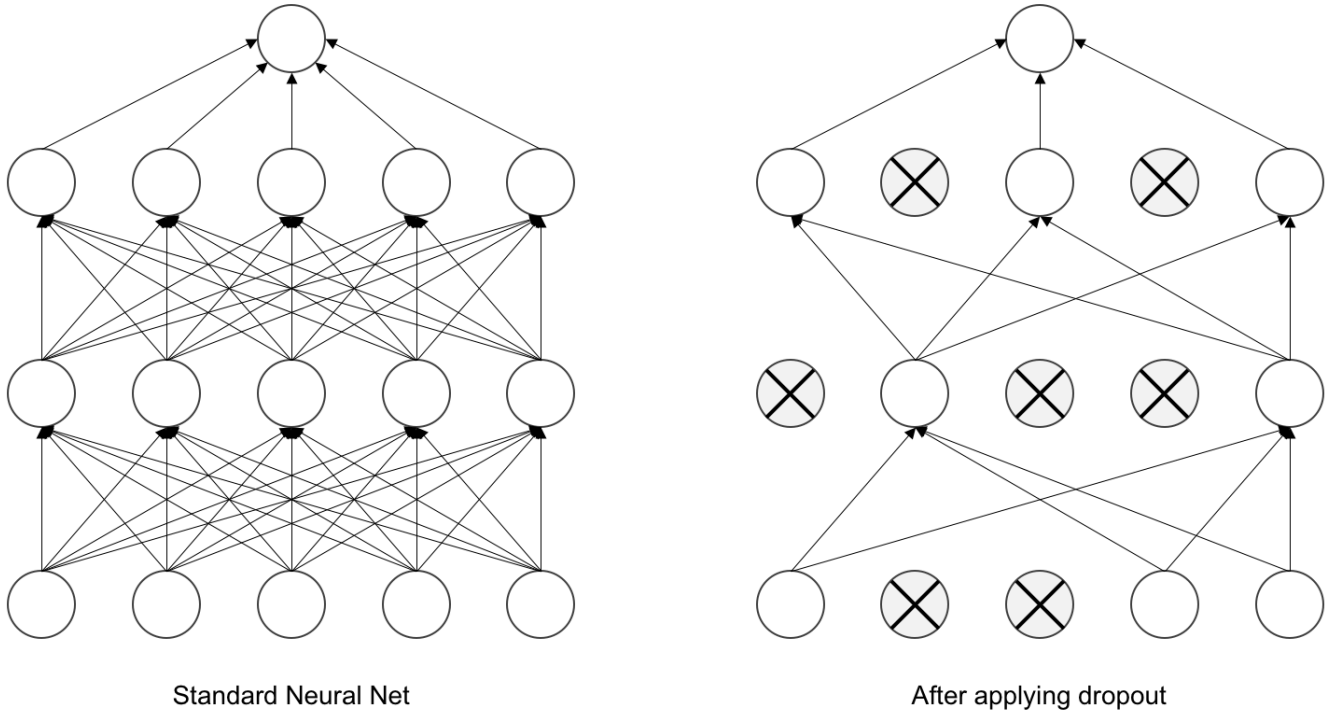
$$w \to w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n} w = \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}$$

Bei jedem Update schrumpft also das Gewicht um den Faktor $1 - \frac{\eta\lambda}{n}$ (weight decay - engl. Gewichtsverfall).

Warum hilft L_2-Regularisierung?

- Ein MLP ist keine SVM: hier wird der Merkmalsraum mitgelernt, somit wird nicht nur die Trennbreite maximiert, sondern zusätzlich auch die Gewichte in den verdeckten Schichten klein gehalten.
- Dennoch eine sinnvolle Strategie: wenn alle Gewichte klein sind, dann ist die Lernmaschine robust gegen Veränderungen an den Gewichten. Um das Verhalten der Maschine zu verändern, müssen viele Gewichte kollektiv angepasst werden. Dies geht nur durch statistisch signifikante Effekte, nicht durch zufälliges Rauschen.
- Die Gewichte können auch auf andere Weise klein gehalten werden, z.B. durch L_1-Regularisierung.

Dropout - Training



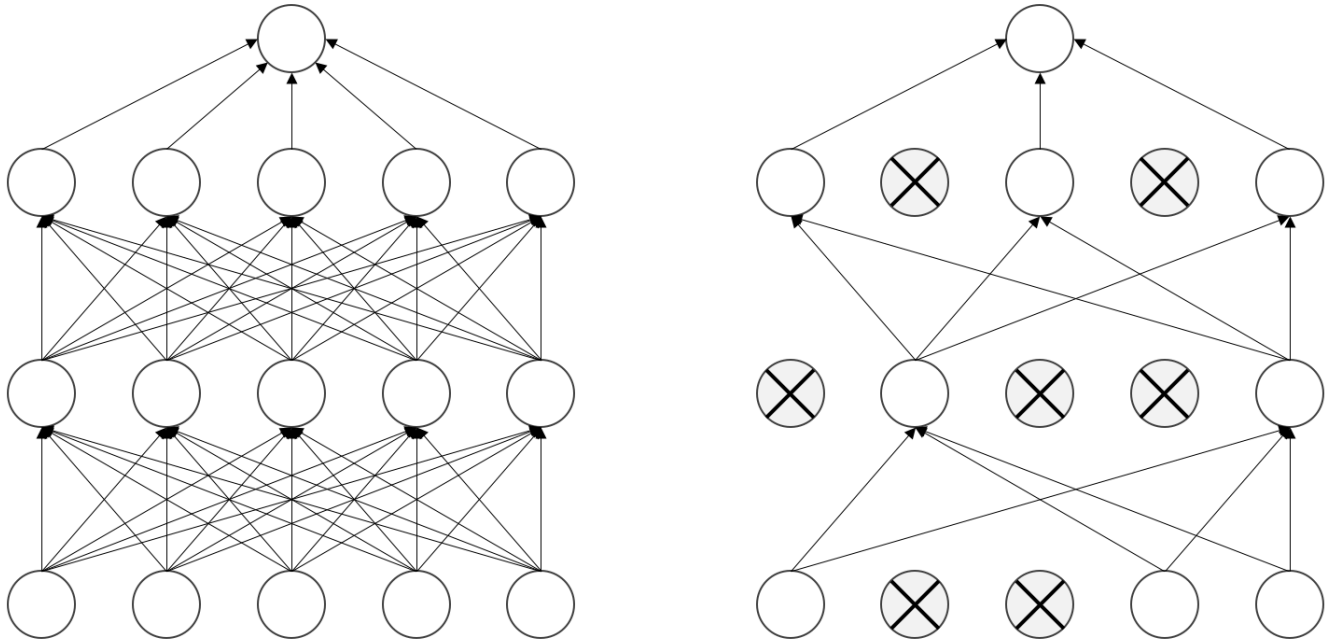
Standard Neural Net

After applying dropout

[Deep Learning for Computer Vision]

Dropout: bei jedem Update wird ein bestimmter Prozentsatz von zufällig gewählten Neuronen abgeschaltet. Dadurch wird in jedem Schritt ein leicht anderes neuronales Netz trainiert.

Dropout - fertig trainiertes Netzwerk



Standard Neural Net

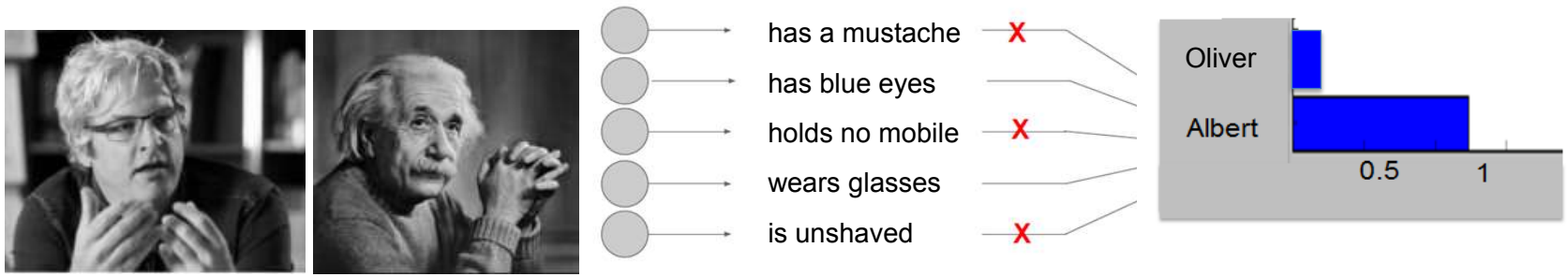
After applying dropout

[Deep Learning for Computer Vision]

Der Output des fertig trainierten Netzwerks ist der Mittelwert aus den einzelnen Unternetzwerken der Trainingsphase. Da im Training nur p Prozent der Neuronen aktiv waren, wird der Gesamtoutput in der Testphase um den Faktor $1/p$ zu hoch sein. Um den gleichen Output wie im Training zu erhalten, muss also der Output mit p multipliziert werden.

Warum hilft Dropout?

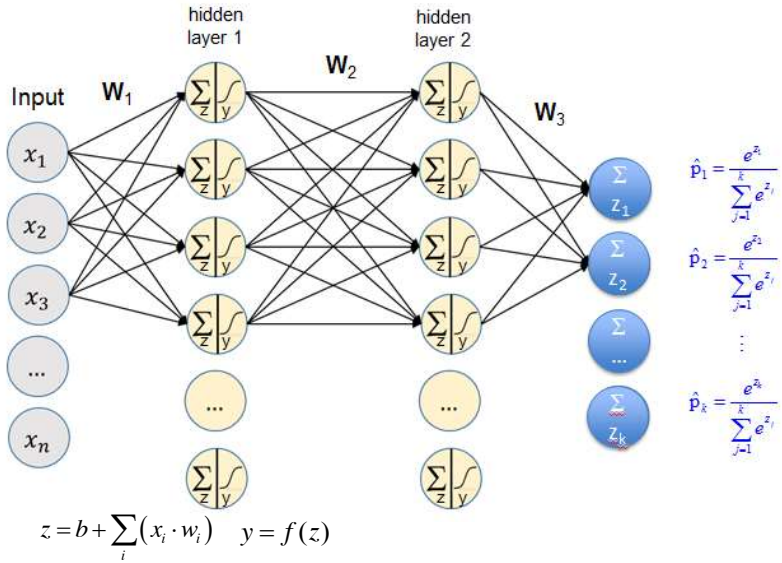
- Netzoutput ist Mittelwert über ein Ensemble (s. Bagging im maschinellen Lernen). Dadurch wird die Genauigkeit und die Robustheit gegenüber Ausreißern erhöht.
- Die Unternetzwerke haben eine einfachere Struktur, was die Gefahr des Overfittings erniedrigt.
- Dropout zwingt das Netzwerk zur Benutzung von redundanten und unabhängigen Merkmalen.



[B.Sick]

Übersicht

Schlechte Idee: konstante Initialisierung aller Gewichte



Vorwärtslauf:
Alle Neuronen einer Schicht haben denselben Output

$$y_j = f(z_j) = f(\sum_i w_i x_i + b),$$

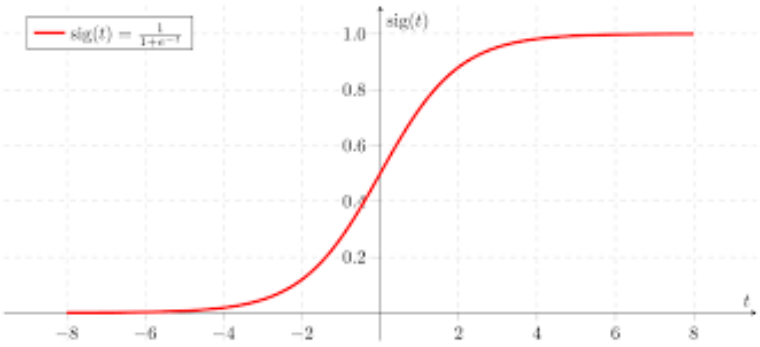
und damit auch das gesamte Netz.

[B. Sick]

Rückwärtslauf:
Alle Gewichte und Neuronen einer Schicht haben dieselben Werte. Damit sind auch alle Gradienten in einer Schicht dieselben. Alle Gewichte bekommen so dieselben Updates und bleiben weiterhin gleich.

Ergebnis: Lernen findet nicht statt.

Noch eine schlechte Idee: Initialisierung mit großen Werten



[Wikipedia]

Backpropagation: Fundamentalgleichungen 1-3:

$$\delta^l = \left((w^{l+1})^\top \delta^{l+1}\right) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

Große Gewichte treiben den Neuronenoutput in die Sättigung und damit in den flachen Bereich der Aktivierungsfunktion. Konsequenz: $\sigma'(z^l)$ ist nahe an 0 und damit auch δ^l, so dass die Gewichte kaum verändert werden.

Auch diese Initialisierung führt zu verlangsamtem Lernen!

Fortpflanzung der Varianzen durch ein neuronales Netz

Angenommen, wir initialisieren alle Gewichte zufällig und unabhängig voneinander mit der Varianz $\text{var}[w]$. Außerdem nehmen wir an, dass alle gewichteten Inputs z^l nahe an 0 sind, so dass näherungsweise $\sigma'(z^l) \approx 1$ und $E[a^{l-1}] = 0$ ist. Für die Varianz des gewichteten Inputs $z^l = w^l a^{l-1} + b^l$ bei n_l unabhängigen neuronalen Inputs mit Varianz $\text{var}[a^{l-1}]$ gilt dann

$$\text{var}[z^l] = n_l \cdot \text{var}[w] \cdot \text{var}[a^{l-1}] \approx n_l \cdot \text{var}[w] \cdot \text{var}[z^{l-1}].$$

Wenn wir wollen, dass die Varianz über die Schichten weder zu- noch abnimmt, so muss gelten:

$$n_l \cdot \text{var}[w] = 1.$$

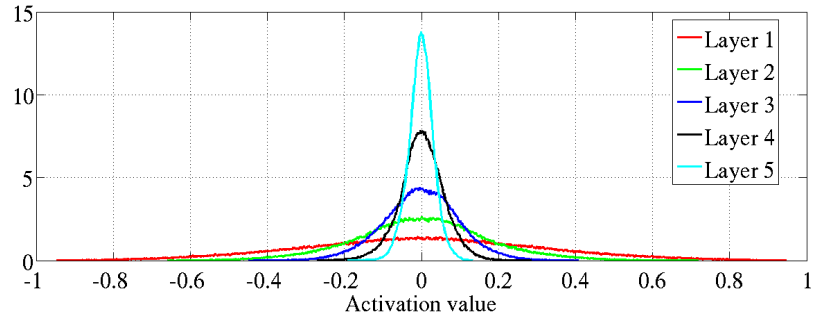
Unter ähnlichen Voraussetzungen gilt für die Varianz des rückpropagierten Fehlers δ^l

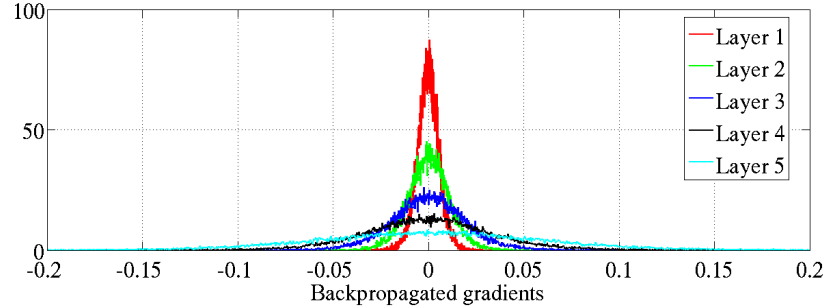
$$\text{var}[\delta^l] \approx n_{l+1} \cdot \text{var}[w] \cdot \text{var}[\delta^{l+1}].$$

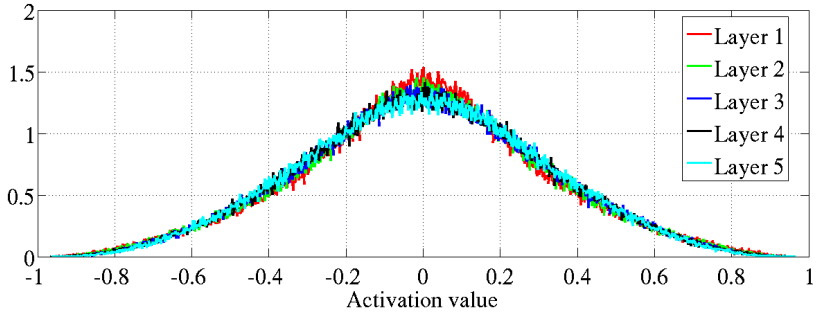
Damit diese über die Schichten gleich bleibt, fordern wir

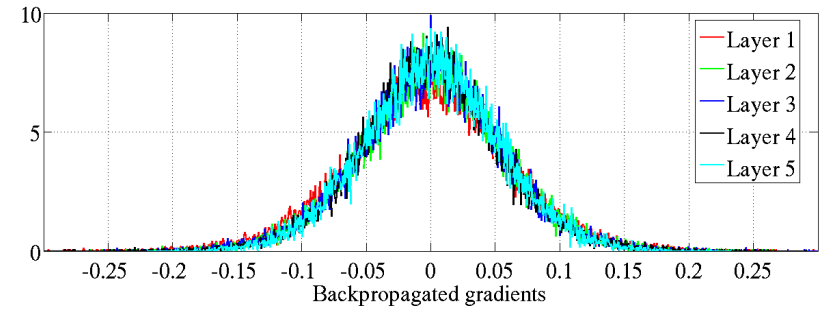
$$n_{l+1} \cdot \text{var}[w] = 1.$$

Initialisierung nach Glorot & Bengio (2010)









Glorot & Bengio (2010) empfehlen bei einer symmetrischen Aktivierungsfunktion als Kompromiss eine Varianz der Gewichte von

$$\mathrm{var}[w] = \frac{2}{n_l + n_{l+1}},$$

z.B. durch eine Gleichverteilung U

$$w \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_l + n_{l+1}}}, \frac{\sqrt{6}}{\sqrt{n_l + n_{l+1}}}\right].$$

Initialisierung nach He et al. (2015)

Bei ReLUs gelten die Annahmen $\sigma'(z^l) \approx 1$ und $E[a^{l-1}] = 0$ nicht, so dass die Initialisierung nach Glorot & Bengio hier nicht geeignet ist. He et al. (2015) berechnen für ReLu die Varianz

$$\text{var}[z^l] \approx \tfrac{1}{2} n_l \cdot \text{var}[w] \cdot \text{var}[z^{l-1}],$$

so dass

$$\tfrac{1}{2} n_l \cdot \text{var}[w] = 1$$

gelten muss. He et al. empfehlen daher eine beschränkte Normalverteilung mit Standardabweichung

$$\sigma = \sqrt{\frac{2}{n_l}}$$

und Mittelwert 0 als geeignete Initialisierung für ReLu-Neuronen.